

01.09.99

Kolster Oy Ab

Iso Roobertinkatu 23

00120 Helsinki

RECEIVED

02 -09- 1999

KOLSTER OY AB

Patentihakemus nro:

982029

Luokka:

H 04Q / JSA / PO

Hakija:

Nokia Telecommunications Oy

Asiamies:

Kolster Oy Ab

Asiamiehen viite:

2980469FI

Määräpäivä

01.03.2000

Patentihakemuksen numero ja luokka on mainittava kirjelmässänne PRH:lle

Patenttivaatimuksissa määritellyt keksinnöt ovat suoritetun tutkimuksen perusteella patentoitavissa (patenttilaki 1 ja 2 §).

Yleistä teknikan tasoa edustaa seuraava julkaisu:

EP 0 709 994; International Business Machines Corporation, 1.5.1999, H04L 29/06

Hakijan tullee toimittaa virastoon PL8§5 momentin mukainen käänös englanninkielellä jätetystä hakemuksesta ja PM38a§3 momentin mukainen vakuutus ennen kuin hakemus tulee julkiseksi.

Tutkijainsinööri  
Puhelin: (09) 69395708

Petri Ojamies

Liitteet: viitejulkaisukopiot ja tutkimusraportti

Lausumanne huomautusten johdosta on annettava viimeistään yllämainittuna määräpäivänä. Jollotte ole antanut lausumaanne virastoon viimeistään mainittuna määräpäivänä tai ryhtynyt toimenpiteisiin tässä välipäätöksessä esitetttyjen puutteellisuksien korjaamiseksi, jätetään hakemus sillensä (patenttilain 15 §). Sillensä jätetty hakemus otetaan uudelleen käsittelväksi, jos Te neljän kuukauden kuluessa määräpäivästä annatte lausumanne tai ryhdytte toimenpiteisiin esitetttyjen puutteellisuksien korjaamiseksi ja samassa ajassa suoritatte vahvistetun maksun, 320 mk hakemuksen ottamisesta uudelleen käsittelväksi. Jos lausumanne on annettu virastoon oikeassa ajassa, mutta esitettynä puutteellisuksia ei ole siten korjattu, että hakemus voitaisiin hyväksyä, se hylätään, mikäli virastolla ei ole aihetta antaa Teille uutta välipäätöstä (patenttilain 16 §). Uusi keksinnön selitys, siihen tehdyt lisäykset ja uudet patenttivaatimuksset on aina jätettävä kahtena kappaleena ja täällöin on otettava huomioon patenttiasetuksen 19 §.

PATENTTIHAKEMUS NRO  982029	LUOKITUS  H04Q7/22, H04L29/06, 29/08
--------------------------------------	--

TUTKITTU AINEISTO	
<p><b>Patenttijulkaisukokoelma (FI, SE, NO, DK, DE, CH, EP, WO, GB, US), tutkitut luokat</b> FI H04L 29/00, 02, 04, 06, 08, 10</p>	
<p><b>Tiedonhaut ja muu aineisto</b> Epoque: Epodoc</p>	

VIITEJULKAISUT		
Kategoria*)	Julkaisun tunnistetiedot	Koskee vaatimuksia
A	EP 0 709 994, IBM Corp., 1.5.1996, H04L 29/06	

\*) X Patentoitavuuden kannalta merkittävä julkaisu yksinään tarkasteltuna  
Y Patentoitavuuden kannalta merkittävä julkaisu, kun otetaan huomioon tämä ja yksi tai useampi samaan kategoriaan kuuluva julkaisu  
A Yleistä tekniikan tasoa edustava julkaisu, ei kuitenkaan patentoitavuuden este

Päiväys  31.8.1999	Tutkija  Petri Ojamies
--------------------------	------------------------------

(19)



Europäisches Patentamt  
European Patent Office  
Office européen des brevets

(11)



EP 0 709 994 A3

(12)

## EUROPEAN PATENT APPLICATION

(88) Date of publication A3:  
29.05.1996 Bulletin 1996/22

(51) Int Cl. 6: H04L 29/06, G06F 13/00

(43) Date of publication A2:  
01.05.1996 Bulletin 1996/18

(21) Application number: 95306980.4

(22) Date of filing: 02.10.1995

(84) Designated Contracting States:  
DE FR GB

• Alvis, Grant Matthew  
Austin, Texas 78727-6735 (US)

(30) Priority: 03.10.1994 US 317980

(74) Representative: Davies, Simon Robert  
IBM  
UK Intellectual Property Department  
Hursley Park  
Winchester, Hampshire SO21 2JN (GB)

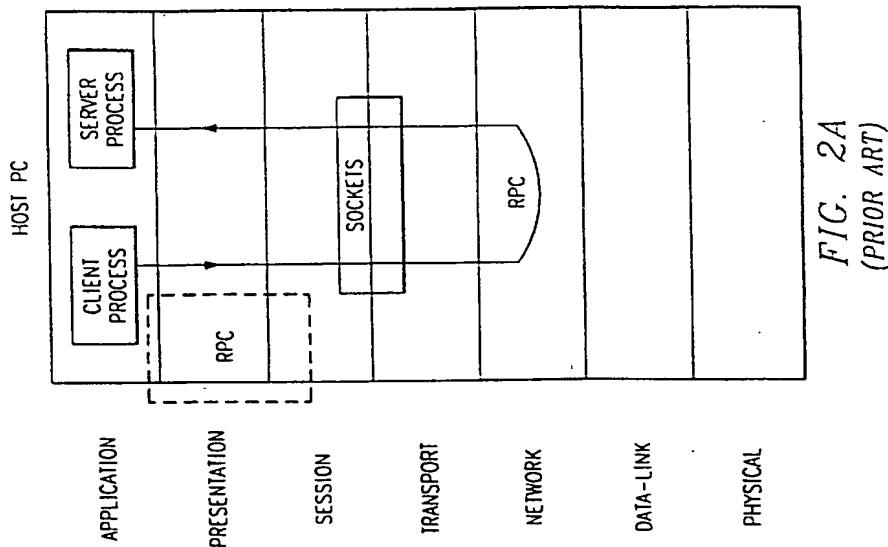
(71) Applicant: International Business Machines  
Corporation  
Armonk, N.Y. 10504 (US)

(72) Inventors:  
• Kapoor, Sandhya  
Austin, Texas 78759 (US)

### (54) Communications management between client and server processes

(57) A method is provided for managing communications between a client process and a server process in a distributed computing environment. The client process resides on a host computer that is connected to a physical network having a transport layer and a network layer. The method begins when the client process makes a remote procedure call by detecting whether a server process identified by the remote procedure call is located on the host computer. If so, a binding handle

vector is returned to the client process, the vector including at least one binding handle having a protocol sequence that establishes an interprocess communication path between the client and server processes instead of a path through the transport and network layers of the physical network. The remote procedure call is then executed, preferably by using a send and receive messaging facility of the host computer operating system.



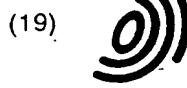
EP 0 709 994 A3



## EUROPEAN SEARCH REPORT

Application Number

DOCUMENTS CONSIDERED TO BE RELEVANT			EP 95306980.4
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl. 6)
A	<u>EP - A - 0 600 235</u> (SOFTWARE AG) * Abstract; claim 1; page 3, lines 3-57; page 6, line 3 - page 8, line 11; fig. 1-3A *	1, 12, 13	H 04 L 29/06 G 06 F 13/00
A	<u>WO - A - 94/06 083</u> (TANDEM COMPUTERS INCORPORATED) * Abstract; page 1, lines 8-34; page 3, line 33 - page 4, line 18; page 7, line 22 - page 12, line 10; fig. 1, 2 *	1, 11-13	
A	<u>WO - A - 94/11 810</u> (MICROSOFT CORPORATION) * Page 1, lines 12-27; page 8, line 25 - page 11, line 3; fig. 3-4C *	1, 12, 13	
A	<u>US - A - 5 249 293</u> (SCHREIBER et al.) * Claim 1; column 1, lines 15-64; column 3, line 23 - column 5, line 6; fig. 1-3 *	1, 12, 13	H 04 L G 06 F
The present search report has been drawn up for all claims			
Place of search	Date of completion of the search	Examiner	
VIENNA	20-03-1996	HAJOS	
CATEGORY OF CITED DOCUMENTS			
X : particularly relevant if taken alone	T : theory or principle underlying the invention		
Y : particularly relevant if combined with another document of the same category	E : earlier patent document, but published on, or after the filing date		
A : technological background	D : document cited in the application		
O : non-written disclosure	L : document cited for other reasons		
P : intermediate document	& : member of the same patent family, corresponding document		



(19)

Europäisches Patentamt

European Patent Office

Office européen des brevets



(11)

EP 0 709 994 A2

(12)

## EUROPEAN PATENT APPLICATION

(43) Date of publication:  
01.05.1996 Bulletin 1996/18

(51) Int Cl. 6: H04L 29/06, G06F 13/00

(21) Application number: 95306980.4

(22) Date of filing: 02.10.1995

(84) Designated Contracting States:  
DE FR GB

- Alvis, Grant Matthew  
Austin, Texas 78727-6735 (US)

(30) Priority: 03.10.1994 US 317980

(71) Applicant: International Business Machines Corporation  
Armonk, N.Y. 10504 (US)(74) Representative: Davies, Simon Robert  
IBM  
UK Intellectual Property Department  
Hursley Park  
Winchester, Hampshire SO21 2JN (GB)

(72) Inventors:

- Kapoor, Sandhya  
Austin, Texas 78759 (US)

## (54) Communications management between client and server processes

(57) A method is provided for managing communications between a client process and a server process in a distributed computing environment. The client process resides on a host computer that is connected to a physical network having a transport layer and a network layer. The method begins when the client process makes a remote procedure call by detecting whether a server process identified by the remote procedure call is located on the host computer. If so, a binding handle

vector is returned to the client process, the vector including at least one binding handle having a protocol sequence that establishes an interprocess communication path between the client and server processes instead of a path through the transport and network layers of the physical network. The remote procedure call is then executed, preferably by using a send and receive messaging facility of the host computer operating system.

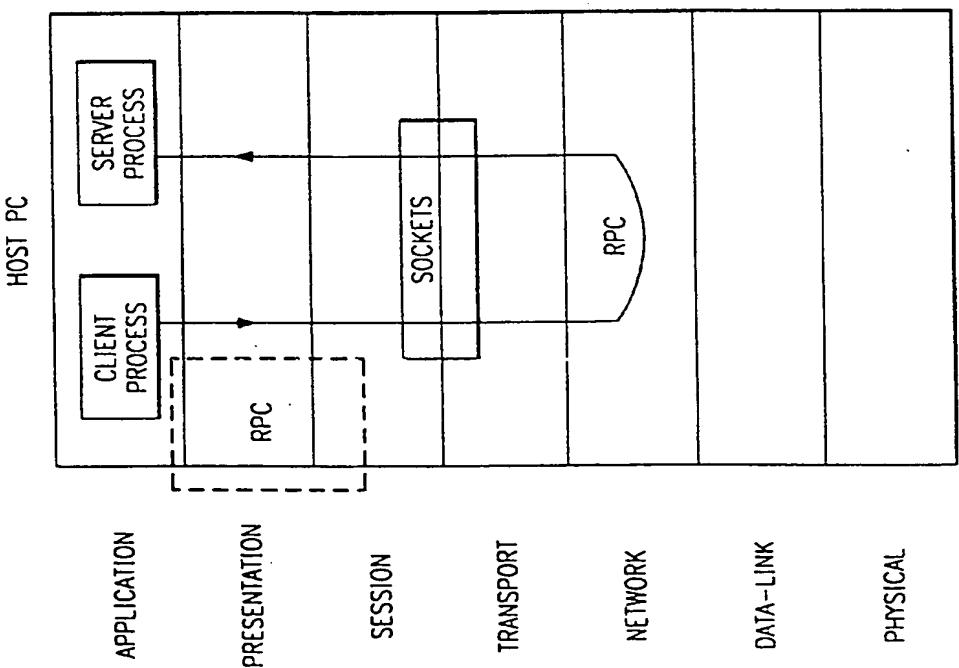


FIG. 2A  
(PRIOR ART)

**Description**

The present invention relates generally to network communications and more particularly to a method for efficiently managing remote procedure calls between client and server processes in a computer network when these processes are supported on the same host computer.

It is well known in the art to interconnect multiple computers into a local area network (LAN) to enable such computers to exchange information and share resources. A local area network provides a distributed computing environment in which users can access distributed resources and process applications on multiple computers. Network communications are carried out using so-called communication protocols. By convention, communication architectures in a local area network are typically characterized as conforming to a seven layer model in the following hierarchy: physical layer, logical link layer, network layer, transport layer, session layer, presentation layer and application layer. The physical layer comprises the actual physical devices and medium used to transmit information. The logical link layer frames data packets and controls physical layer data flow, insuring delivery of data regardless of the actual physical medium. The network layer addresses and routes data packets. It creates and maintains a route in the network between a source node and a destination node. The transport layer creates a transport pipeline between nodes and manages the network layer connections. The session layer typically provides remote procedure call (RPC) support, maintains the integrity of the connection between nodes and controls data exchange. The presentation layer encodes and decodes data and provides transparency between nodes. Finally, the application layer provides the interface to end-user processes and provides standardized services to applications.

The seven layer model has many variations depending on the particular network architecture. Thus, for example, in a network architecture based on the TCP/IP (Transmission Control Protocol/Internet Protocol) interface running IBM RISC System/6000 computer workstations under the AIX Operating System, there is another layer, called the socket layer, that sits between the session and transport layers. The socket layer creates so-called "sockets" which are logical constructs analogous to physical ports. In this architecture, the RPC mechanism is not just supported in the session layer, but also includes functionality of the session layer. A known RPC mechanism useful in distributed computing environments (DCE) includes software code provided by the Open Systems Foundation (OSF).

The OSF DCE RPC mechanism is used conventionally to manage communication between a "client" and a "server" in a distributed computing environment, with the client requesting a service from a server using a remote procedure call (RPC). As is well known, a "client" refers to a network participant that is requesting a service accessible somewhere within the computing environment. A "server" provides the requested service to a client. With the OSF DCE RPC mechanism, each client process (namely, a process running on a client machine) has an associated socket created by the socket layer. Each server process likewise is associated with a socket. In response to an RPC, a call directory service returns a data structure, called a "binding handle," specifying the location of the server process as a network address and the port number where the server process is running. The binding handle is then used by the RPC mechanism to define a communication path between the client process and the server process. The path is defined using ip-based (i.e., network layer) protocol sequences of the Internet Network Address Family (AF\_INET) to open the sockets. The path loops down from the client process through the transport and network layers, out on the network and then back up the layers associated with the host on which the server process is running.

The OSF DCE RPC mechanism as described above cannot distinguish whether client and server processes are running on the same host machine. In all cases, the mechanism returns a binding handle to the client process including an AF\_INET protocol sequence that sets up a communication path through the transport (TCP or UDP) layer and the network (IP) layer. Communications through TCP use connection-oriented protocol sequences while those through UDP use connection-less protocol sequences, but in either case, when the client and server processes reside on the same host, an RPC generates a so-called loopback message because once the network (IP) layer receives the destination network address, it recognizes that the RPC is "local"; the path must therefore be looped back up through the transport layer to the server process on the application layer. Because of this loopback requirement, RPC's between client and server processes on the same machine are not optimized from a performance standpoint as they use the transport and network layers unnecessarily.

Accordingly the invention provides a method for managing communication between a client process and a server process in a distributed computing environment, the client process residing on a host computer that is connected to a physical network having a transport layer and a network layer, comprising the steps of:

- (a) when a remote procedure call is made by the client process, detecting whether a server process identified by the remote procedure call is located on the host computer;
- (b) if the server process is located on the host computer, establishing an interprocess communication path between the client process and the server process without use of the transport and network layers of the physical network; and
- (c) executing the remote procedure call.

a connection-oriented protocol sequence being used to open such sockets based on the UNIX Network Address Family (AF\_UNIX). With AF\_UNIX, the operating system kernel handles the task of bridging communications between the two processes on the same host.

A novel protocol sequence may be employed as a data structure useful for improving local RPC performance. The data structure preferably includes a unique identifier so that a unique socket file is used for each association established between a client and server process. This ensures that a socket file will not be used again on multiple invocations of a DCE application.

It is important that such a protocol sequence as described above does not cause any compatibility or interoperability problems with existing DCE applications. Therefore, applications using previous versions of the DCE shared library may be allowed to communicate with applications using a DCE shared library supporting the protocol sequence. Implementation of communications management in accordance with the invention may therefore be substantially invisible to the user and backwards compatible with existing client/server applications.

Thus a method is provided for managing communications between a client process and a server process in a distributed computing environment, with the client process residing on a host computer that is connected to a physical network having a transport layer and a network layer. The method begins when a client process makes a remote procedure call (RPC) by detecting whether a server process identified by the remote procedure call is located on the host computer. If so, typically a binding handle vector is returned to the client process including at least one binding handle having a protocol sequence that establishes an interprocess communication path between the client and server processes instead of a path through the transport and network layers of the physical network. The remote procedure call is then executed, preferably by using a send and receive messaging facility of the host computer operating system.

Various preferences may be established to ensure that local RPC's use the novel protocol sequence if at all possible. Thus the binding handle vector may be returned to the client process with handles having the new protocol sequence. If the binding handle vector is also returned with ip-based protocol sequences, the binding handles with the new protocol sequence are used before any binding handle with an ip-based protocol sequence.

An embodiment of the invention will now be described in detail by way of example only with reference to the following drawings:

FIGURE 1 schematically illustrates a computer network;

FIGURE 2A illustrates a conventional RPC using the network and transport layers of the physical network;

FIGURE 2B illustrates a "local" RPC carried out using an IPC mechanism of the host computer in accordance with the present invention;

and FIGURE 3 illustrates a preferred method for managing local remote procedure calls using preferences to bias the RPC mechanism to select AF\_UNIX protocol sequences over ip-based protocol sequences.

A known distributed computing environment (DCE) is illustrated in FIGURE 1 and includes clients 10 interconnected to servers 12 via a network 14. Each of the clients 10 and servers 12 is a computer. For example, each computer may be an IBM RISC System/6000 workstation running the AIX operating system. The AIX operating system is compatible at the application interface level with AT&T's UNIX operating system, version 5.2. The various models of the RISC-based personal computers are described in many publications of the IBM Corporation, for example, RISC System/6000, 7073 and 7016 POWERstation and POWERserver Hardware Technical Reference, Order No. SA23-2644-00. The AIX operating system is described in AIX Operating System Technical Reference, published by IBM Corporation, First Edition (November, 1985), and other publications. A detailed description of the design of the UNIX operating system is found in a book by Maurice J. Bach, Design of the Unix Operating System, published by Prentice-Hall (1986).

In one particular implementation, a plurality of IBM RISC System/6000 are interconnected by IBM's System Network Architecture (SNA), and more specifically SNA LU 6.2 Advanced Program to Program Communication (APPN). SNA uses as its link level Ethernet, a local area network (LAN) developed by Xerox Corporation, or SDLC (Synchronous Data Link Control). A simplified description of local area networks may be found in a book by Larry E. Jordan and Bruce Churchill entitled Communications and Networking for the IBM PC, published by Robert J. Brady (a Prentice-Hall Company) (1983).

A different implementation uses IBM's PS/2 line of computers running under the OS/2 operating system. For more information on the PS/2 line of computers and the OS/2 operating system, the reader is directed to Technical Reference Manual Personal Systems/2 Model 50, 60 Systems IBM Corporation, Part No. 68x2224 Order Number S68X-2224 and OS/2 2.0 Technical Library, Programming Guide Volumes 1-3 version 2.00, Order Nos. 10G6261, 10G6495 and 10G6494.

It will be appreciated that the teachings herein may be implemented using a wide variety of different computers

ncacn\_unix\_stream:[/tmp/6094]  
 ncacn\_unix\_stream:[/5423]

Note in the last case, because the endpoint is not represented by an absolute path, that the ncacn\_unix\_stream: [/var/dce/rpc/socket/5423] is assumed, and will be used to create the socket file. The entry in the endpoint map will also have an absolute path. This will not cause any problems when using the two forms interchangeably, because the relative path will always be expanded out to an absolute path before it is ever used.

The more detailed characteristics of the ncacn\_unix\_stream protocol sequence can now be described. The RPC runtime library recognizes when the client and server are on the same host, and will "encourage" the use of ncacn\_unix\_stream, when it is appropriate, in the following ways:

- \* rpc\_ns\_binding\_import\_next() will return all of the ncacn\_unix\_stream bindings before it will return bindings for other protocol sequences;
- \* rpc\_ns\_binding\_lookup\_next() will return all of the ncacn\_unix\_stream bindings in the lower positions of a binding vector that gets returned. For example, if a returned binding vector consists of 5 binding handles, and 2 of them are ncacn\_unix\_stream, then the ncacn\_unix\_stream bindings would occupy array elements [0] and [1], and the bindings for other protocol sequences would occupy the remaining elements, [2], [3], and [4]. Note that although the ncacn\_unix\_stream bindings will be in the lower elements of the binding vector, the ordering of the remaining binding handles is indeterminate;
- \* rpc\_ns\_binding\_select() will always select the ncacn\_unix\_stream binding handles from a binding vector, until they have all been used. It will then resume its normal policy of randomly selecting amongst the binding handles of the other protocol sequences.

The above-mentioned preferences that the RPC runtime gives to ncacn\_unix\_stream has the net effect of giving DCE applications a performance boost. Existing applications do not even need to be aware of these preferences. They will be virtually invisible to the application, and do not require any changes to existing programs.

With reference now to FIGURE 3, a method for managing RPCs that implement some of these preferences is illustrated. The method manages communication between client and server processes in a distributed computing environment, with the client process requesting a service from a server process using an OSF DCE remote procedure call (RPC). The client process resides on a host computer that is connected to a physical network having a transport layer and a network layer. Preferably, the host computer supports a UNIX-based operating system.

The method begins at step 30 by detecting whether a server process identified by a remote procedure call is located on the host computer. If so, client/server 15 of FIGURE 1 is being used, and at step 32 the RPC mechanism of FIGURE 2A returns to the client process a vector of binding handles including a first set and a second set. Each of the first set of binding handles includes a protocol sequence that establishes an interprocess communication path between the client process and the server process without use of the transport and network layers, with each of the second set of binding handles including a protocol sequence that establishes a communication path between the client process and the server process through use of the transport and network layers. At step 34, an attempt is made to execute the remote procedure call using one of the binding handles in the first set. If this fails, a test is performed at step 36 to determine whether all of the binding handles in the first set have been used. If the result of the inquiry is negative, the routine loops back and attempts to execute the RPC with another ncacn\_unix\_stream" protocol sequence. If the result of the test at step 36 is positive, indicating that all such protocol sequences have been used, the method continues at step 38 to use one of the binding handles in the second set to attempt to execute the remote procedure call. The binding handles in the second set are accessed randomly.

The conditions under which the ncacn\_unix\_stream protocol sequence will not be utilized are:

- \* A DCE server application explicitly specifies the support of a particular protocol sequence. An example would be the use of rpc\_server\_use\_protseq() by a server to limit the protocol sequences it supports to be a subset of all protocol sequences available.
- \* A DCE client application chooses from amongst the bindings returned from the namespace by comparing against a particular component of the binding. An example would be comparing-against the network address of the binding. Because ncacn\_unix\_stream bindings contain no network address, this comparison would never succeed (unless the client application was comparing against the "" string). Another example would be the case where a check is done on each binding, and the client is looking for a particular protocol sequence. Both of these examples are normally carried out by using the rpc\_binding\_to\_string\_binding() routine, and then comparing the string format of the components that make up the binding.

APPENDIX5      **0.1.1 External Interface Descriptions**

The external interface of RPC runtime and RPCD is changed as follows:

10     The user has the new ability to specify an additional protocol sequence in addition to those already supported. The new protocol sequence is named "ncacn\_unix\_stream". Anywhere that an API function parameter, environmental variable, or command line argument allows for the specification of the string format for a protocol sequence, the string "ncacn\_unix\_stream" will be allowed.

15      **0.1.2 Internal Design Description**

15     The RPC enhancement adds support for a new protocol sequence (ncacn\_unix stream), that is used when a client and server exist on the same host machine.

20     This RPC enhancement is virtually invisible to the user, and is backwards compatible with existing client/server applications.

25     The invention is portable to other platforms, such as the IBM OS/2 platform (note that OS/2 has support for the UNIX Domain, via MPTS, and the Common Porting Platform (CPP) DLLs on OS/2 will allow almost all of the UNIX specific subroutine calls to be used without modification on OS/2).

25     The design of the RPC runtime library is such that the addition of a new protocol sequence is accomplished with very little modification to the existing code. This is achieved via two means: 1) Modular data structures, and 2) The use of entry point vectors. Modular data structures facilitate the effort of adding the new protocol sequence by providing a consistent mechanism by which to relate the various components that make up the protocol sequence. The majority of the RPC runtime code is common to all protocols, so the use of entry point vectors allows protocol specific routines to be accessed in-line, via an index into an array of functions. The protocol id is used as the index into the entry point vector. New protocol sequences require the addition of new routines to handle processing that is specific to that protocol sequence.

35     The ncacn\_unix\_stream protocol sequence is built using the AF\_UNIX address family and a socket type of SOCK\_STREAM. The socket address uses the structure sockaddr\_un, using a complete path name specification as the interprocess communication mechanism. The client and server must reside on the same host to use this protocol sequence. The protocol sequence is connection-oriented, and uses the CN protocol id.

40     The rest of this section of the document discusses the RPC source files that require modification in order to implement the new protocol sequence. An attempt has been made to list the files, starting with wide implication changes, and narrowing down to more specific changes.

45      **0.1.2.1 src/rpc/runtime/com.h**

50     This is an existing file. The addition of the following constant identifiers are necessary. These constants are used throughout the RPC runtime code to access appropriate structures and properly index entry point vectors.

50     RPC protocol sequence id constant:

```
#define rpc_c_protseq_id_ncacn_unix_stream 5
```

55     RPC protocol sequence string constant:

```
#define rpc_protseq_ncacn_unix_stream "ncacn_unix_stream"
```

```

5      rpc_naf_set_pkt_nodelay_fn_t naf_set_pkt_nodelay;
      rpc_naf_is_connect_closed_fn_t naf_is_connect_closed;
      rpc_naf_twr_flrs_from_addr_fn_t naf_tower_flrs_from_addr;
      rpc_naf_twr_flrs_to_addr_fn_t naf_tower_flrs_to_addr;
      rpc_naf_desc_inq_peer_addr_fn_t naf_desc_inq_peer_addr;
} rpc_naf_epv_t, *rpc_naf_epv_p_t;

Each NAF has an initialization routine that loads up this structure, and
adds it to the rpc_g_naf_id[] table entry for the specified NAF. From
that point on, all calls to the Unix Network Address Family are made
through these EPVs.

10     The following routines must be created to implement the Unix Network
      Address Family.

15     0.1.2.3.1 rpc_unix_init()

16     PRIVATE void rpc_unix_init(naf_epv,status)
17         rpc_naf_epv_p_t*naf_epv;
18         unsigned32*status;
19         {

20             Load the rpc_unix_epv structure with the static routines defined in
this file (unixnaf.c).
              Return the NAF epv, so it can be added to the global NAF table.

21         }

22     0.1.2.3.2 addr_alloc()

23     INTERNAL void addr_alloc(rpc_protseq_id_naf_id, endpoint, netaddr, network
options, rpc_addr, status)
24         rpc_addr_p_t src_rpc_addr;
25         rpc_addr_p_t*dst_rpc_addr;
26         unsigned32*status;

27     {

28         Allocate memory for the destination RPC address
         Copy source rpc address to destination rpc address
29     }

30     0.1.2.3.4 addr_free()

31     INTERNAL void addr_free (rpc_addr,status)
32         rpc_addr_p_t *rpc_addr;
33         unsigned 32 *status;
34         {

35             Free memory of RPC addr and set the RPC address pointer to NULL.
36         }

37     0.1.2.3.5 addr_set_endpoint()

38     INTERNAL void addr_set_endpoint (endpoint, rpc_addr,status)
39         unsigned_char_p_t endpoint;
40         rpc_addr_p_t *rpc_addr;
41         unsigned32 *status;
42         {

43             Check for the special case where endpoint is a pointer to a zero
length string (as opposed to NULL). In this case, the caller is
requesting that the endpoint be zeroed out of the rpc_addr
structure, so just set rpc_addr->sa.sun_path[0] = '\0' and return.

44         BEGIN AIX ONLY
45             Check for the existence of the RPC_UNIX_DOMAIN_DIR_PATH environment
variable.
46             if it exists then use it as the base for file name paths.
47             if it doesn't exist, then use the default: /var/dce/rpc/socket on
AIX.
48
49
50
51
52
53
54
55

```

```

This is a no-op routine. There is no network options concept when
using the Unix Domain.
The routine must exist though, for consistency with other NAFs that
do require it.

5
}

0.1.2.3.10 addr_inq_options()

10 INTERNAL void addr_inq_options (rpc_addr, network_options, status)
    rpc_addr_p_t rpc_addr;
    unsigned_char_t **network_options;
    unsigned 32 *status;
{
    This is a no-op routine. There is no network options concept when
    using the Unix Domain.
    The routine must exist though, for consistency with other NAFs that
    do require it.
}

15

0.1.2.3.11 inq_max_tsdu()

20 INTERNAL void inq_max_tsdu (naf_id, iftype, protocol, max_tsdu, status)
    rpc_naf_id_t naf_id;
    rpc_network_if_id_t iftype;
    rpc_network_protocol_id_t protocol;
    unsigned 32 *max_tsdu;
    unsigned 32 *status;
{
    This is a no-op routine. The packets will not be going out on the
    network, so IP settings are not relevant.
}

25

0.1.2.3.12 addr_compare()

30 INTERNAL boolean addr_compare (addr1, addr2, status)
    rpc_addr_p_t addr1, addr2;
    unsigned 32 *status;
{
    Compare the socket address file name paths of the 2 RPC addrs
    passed in.
    Return TRUE or FALSE.
}

35

0.1.2.3.13 inq_max_pth_unfrag_tpdu()

40 INTERNAL void inq_max_pth_unfrag_tpdu
    (rpc_addr, iftype, protocol, max_tpdu, status)
    rpc_addr_p_t rpc_addr;
    rpc_network_if_id_t iftype;
    rpc_network_protocol_id_t protocol;
    unsigned32 *max_tpdu;
    unsigned 32 *status;

    This is a no-op routine. The packets will not be going out on the
    network, so IP settings are not relevant.
}

45

0.1.2.3.14 inq_max_loc_unfrag_tpdu()

50 INTERNAL void inq_max_loc_unfrag_tpdu
    (naf_id, iftype, protocol, max_tpdu, status)
    rpc_naf_id_t naf_id;
    rpc_network_if_id_t iftype;
    rpc_network_protocol_id_t protocol;
    unsigned32 *max_tpdu;
    unsigned32 *status;
{
}

```

```

        Add the socket address to an RPC address and return it.
    }

5   0.1.2.3.20 desc_inq_peer_addr()

INTERNAL void desc_inq_peer_addr (protseq_id, desc, rpc_addr, status)
rpc_protseq_id_t protseq_id;
rpc_socket_t desc;
rpc_addr_p_t *rpc_addr;
unsigned32 *status;

10  {

    Allocate memory for the new RPC address, and fill in the protseq id
    and length of the sockaddr structure.

    Call rpc_socket_inq_endpoint(), which will fill in the peer
    endpoint, which is always the same as the current processes endpoint, in
    the case of Unix Domain.
}

```

#### 0.1.2.4 src/rpc/runtime/unixnaf.h

20 This is a new file that must be created. This file primarily contains prototypes for the Unix NAF epv routines. In addition, it contains a definition for the representation of a Unix Domain RPC address. It is defined as follows:

```

25 typedef struct rpc_addr_unix_t
{
    rpc_protseq_id_t rpc_protseq_id;
    unsigned32 len;
    struct sockaddr_un sa;
} rpc_unix_addr_t, *rpc_unix_addr_p_t;

```

30 When RPC is executing command code, the RPC address is passed around as a quasi-opaque structure (rpc\_addr\_t), but this structure is cast to the RPC address structure for the appropriate family when it is passed to a NAF specific routine; in the case of unix stream, it is cast to rpc\_unix\_addr\_t.

35 Another necessary addition to this file is the default pathname to be used for creating filenames for use with Unix Domain socket calls on AIX. This is Operating System specific, and for example is not necessary on OS/2. When creating an endpoint, if the endpoint does not exist, or the endpoint to be used does not specify a full pathname, then the default pathname is used. The default pathname will be defined as follows:

```

40 #ifdef(AIX_PROD)
#define RPC_DEFAULT_UNIX_DOMAIN_PATH "/var/dce/rpc/socket"
#endif

```

45 On the OS/2 operating system, Unix Domain socket files are only used internally. No user viewable file gets created. The OS/2 operating system handles the administrative tasks associated with the socket file (ie cleanup). No special path generation is necessary.

#### 0.1.2.5 src/rpc/runtime/RIOS/unixnaf\_sys.c

50 This is a new file that must be created. It contains routines that are system specific, as pertains to the Unix Domain Network Address Family.

##### 0.1.2.5.1 rpc\_unix\_desc\_inq\_addr()

55 Receive a socket descriptor which is queried to obtain family, endpoint and network address. If this information appears valid for a Unix Domain address, space is allocated for an RPC address which is initialized with the information obtained from the socket. The address indicating the created RPC address is returned in rpc\_addr.

will be empty, since it is just a place holder, and is not used for anything.

5           }

### 0.1.2.7.2 twr\_unix\_lower\_flrs\_to\_sa()

Creates a Unix Domain sockaddr from the canonical representation of a Unix Domain protocol tower's lower floors.

10         PUBLIC void twr\_unix\_lower\_flrs\_to\_sa (tower\_octet\_string,sa,sa  
len,status)  
byte\_p\_t tower\_octet\_string;  
sockaddr\_p\_t \*sa;  
unsigned32\*sa\_len;  
unsigned32\*status;  
15          {  
               Skip over the upper 3 floors of the tower reference, and position  
               to the lower network floors.  
  
               Sequence thru the tower octet string, and build a unix socket  
               address. This is the converse of the routine twr\_unix\_lower\_flrs  
               from\_sa().  
20          }  
25          }

### 0.1.2.7.3 rpc\_rpcd\_is\_running()

This is a new routine. It is used for two purposes: 1) the RPCD will use it during initialization to determine if an instance of RPCD is already running, and 2) the RPC runtime will use it to determine if the RPCD on the local host is running, and if it supports the ncacn\_unix\_stream protocol sequence.

30         This routine checks for an instance of RPCD that supports ncacn\_unix stream. This is done by looking for the existence of the rpcd endpoint file (/var/dce/rpc/socket/135). If the file does not exist, then no RPCD is running with support for Unix Streams. If the file does exist, it uses it to attempt to connect to the RPCD instance. If the connect fails, then no RPCD is running with support for Unix Streams. Remove the file. If the connect is successful, then RPCD is running. Do not remove the file.

35         PRIVATE boolean32 rpc\_rpcd\_is\_running(status)  
               unsigned32 \*status;  
               {  
               Check if the Unix Streams protocol sequence is supported. If not,  
               return the error rpc\_s\_protseq\_not\_supported.

40         Inspect the interface specification for the RPCD (ept\_v3\_0\_cifspec)  
               and get the well-known endpoint for the Unix Streams protocol  
               sequence.

45         The endpoint is used as a socket file for communicating to the  
               RPCD. First, check if the file exists. If it does exist, then we  
               expect it to be a file associated with a socket, and the fopen()  
               should fail with the appropriate error. If it doesn't exist, then  
               there is no rpcd running (at least with support for ncacn\_unix  
               stream). So just return. Everything is Ok.

50         If this file can be opened, then it is NOT a socket file.  
               Therefore remove it (i.e. it shouldn't be therein the first place)  
               and return.

55         If the file exists, and is a socket file, check if there is an RPCD  
               actively using it. Do this via a socket()/connect() sequence.

If no rpcd is running, we expect the connect call to fail. We  
check for the proper error, and flag the case where an unexpected  
error occurs.

```

5
    {
        rpc_list_t link;
        char *entry_name;
        rpc_binding_handle_t b_handle;
        rpc_if_rep_p_t if_spec;
        uuid_t obj_uuid;
    }ep_lkup_t, *ep_lkup_p_t;

10   0.1.2.11 nslookup.c

15   This is an existing routing. It returns a list of binding handles of one
    or more compatible servers (if found) from the name service database.
    This routing must be modified so that the ncacn_unix_stream binding
    handles associated with each unique namespace entry that gets resolved in
    the lookup are prepended to the binding vector that gets returned to the
    user. This means that the lookup proceeds as normal, and then, just
    before returning to the user, the routine makes a call to see if there
    are any ncacn_unix_stream bindings in the endpoint map that match the
    object uids and interface ids of the bindings that are in the binding
    vector. If there are, then they are prepended to the binding vector, and
    the concatenated vector is given to the user. The following return
    statements in rpc_ns_binding_lookup_next() need to be modified to get
    local bindings before returning:
20

25   NOTE: See below for the details of the routine rpc_prepend_local
    bindings().

30   ..../* code not of interest to us */
.....

35   case rpc_s_priority_group_donbe:
        rpc_next_priority_group_count(lookup)_node);
        if((binding_vector)->count>0)
        {
            *status = rpc_s_ok;
            rpc_prepend_lcl_bndgs(binding_vector, lookup_rep, status);
            return;
        }

40   case rpc_s_binding_vector_full:
        *status=rpc_s_ok;
        rpc_prepend_lcl_bndgs(binding_vector, loopup_rep, status);
        return;

45   ..../* code not of interest to us */
.....
/*
 * if any compatible bindings were found, return them
 * before processing any other attributes on this entry
 */
if ((binding_vector)->count>0)
{
    *status = rpc_s_ok;
    rpc_prepend-lcl_bndgs (binding_vector, lookup_rep, status);
    return;
}

50   ..../*code not of interest to us */
.....
/*
 *if we're done with the node list, see what's in the binding
 *vector
 */

```

```

get the head element, and using its interface specifier and object
uuid, determine the inquiry type for making the endpoint map
inquiry.

5      call rpc_mgmt_ep_elt_inq_begin()

DO
    call rpc_mgmt_ep_elt_inq_next()
    If the protocol sequence for the returned element is ncacn
    unix_stream
        break out of the do loop
    Endif
10     While the status is OK

    call rpc_mgmt_ep_elt_inq_done()

15     If there was a uuid used to qualify the endpoint inquiry
          add the object to the binding handle (rpc_binding_set
          object())
    Endif

20     call rpc_binding_reset() to get rid of the endpoint. We only want
          a partially bound handle, so it will be consistent with the other
          binding handles in the binding vector. Also, since we only do the
          endpoint map inquiry until we find one unix stream binding, we
          don't want the endpoint because it would be deterministic as to
          what the binding endpoint would be. By resetting the binding,
          and forcing a call to resolve the binding, we are making the
          selection non-deterministic, or random, which is what we want.

25     add the binding to the new binding vector

Endwhile

If any unix stream bindings were found in the endpoint map
30     copy the original binding vector elements to the end of the newly
          formed vector
     free the original binding vector
     assign the newly formed binding vector to the return parameter for
          the binding vector.
endif

35     NOTE: If no elements were found, then the original binding vector is
          returned, with nothing altered.
]

0.1.2.11.3 rpc_ns_binding_select()

40     This is an existing routine. It randomly selects a binding handle from a
          vector of binding handles passed into the routine. This routine is
          modified so that the unix stream binding handles will be chosen first.
          After all unix stream binding handles have been chosen, then the
          remaining binding handles (representing other protocol sequences) will be
          chosen randomly. In the event that there is more than one unix stream
45     binding handle, the routine should randomly select from amongst them.
          the algorithm goes like this:

..../* Determine that there are still unused bindings in the binding
vector */

50     .....

55     If ncacn_unix_stream protocol sequence is supported
          For each binding in the binding vector
              If the protocol sequence for this binding is ncacn_unix
                  stream
                      select the binding
                      NULL out the binding vector element
                      return

```

## 0.1.2.13.2 rpc\_local\_host\_support\_only()

5 This is a new routine. It queries the global protocol sequence table, and determines if ncacn\_unix\_stream is the only supported protocol sequence. It returns true or false. This routine is only called from places where it is known that ncacn\_unix\_stream is supported, so no check is necessary as to whether ncacn\_unix\_stream itself is supported. The code is as follows:

```
10    For each element in the rpc_g_protocol_id table
        if the protocol is supported and the protocol is not ncacn_unix
        stream
            return FALSE
        endif
        return TRUE
    endfor
```

## 15 0.1.2.14 src/rpc/sys\_idl?ep.ids

20 The file src/rpc/sys\_ids/ip.ids contains the RPC interface used to access the RPCD endpoint database. This idl file contains the well-known endpoint that RPCD uses everytime it runs. The endpoint is specified in this file on a per-protocol sequence basis, so an addition must be made to this file for the ncacn\_unix\_stream protocol sequence as follows:

```
25    [
        uuid(e1af8308-5d1f-11c9-91a4-08002b14a0fa),
        version(3.0),
        endpoint( "ncadg_id_udp:[135]",
                  "ncadg_dds:[12]",
                  "ncacn_ip_tcp:[135]",
                  "ncacn_dnet_nsp:[#69]",
                  "ncacn_unix_stream"]135),
        pointer_default(ptr)
    }

30    interface ept
    {
        .../* Rest of idl specification */
    }
```

35 Note that the endpoint will be converted into a filename, using the rules given in the routing rpc\_addr\_set\_endpoint(), described previously in this document. This means the 135 will be prepended with the path set in the RPC\_DEFAULT\_UNIX\_DOMAIN\_PATH definition (see description of file "tvr unix.c"), depending on the platform in use.

## 40 0.1.2.15 src/rpc/rpcd/rpcd.c

45 This is an existing file. This file needs to be modified so that when the rpcd server comes up, it checks if the socket file that is used to listen for requests over ncacn\_unix\_stream exists. If it does, then it needs to delete it. This is because a socket file cannot be created if the file already exists.

## 45 0.1.2.15.1 main ()

The following code must be added after the database has been initialized, but before the protocol sequence support is set up:

```
50    ..../* database initialization */
    ...

    if (rpc_rpcd_is_running(&status))
    {
        fprintf(stderr,
                "(rpcd) Instance already running. Can't continue.\n");
        exit(1);
```

```

for (i = 0; i<psvp->count; i++)
{
    printf("%s/n", psvp->protseq[i]);
}

```

### 0.1.2.19 Installp Utility

The install image for DCE is enhanced so that it creates (or overwrites) the directory /usr/tmp/rpc. Without this directory, DCE as well as any DCE applications will not work over ncacn\_unix\_stream.

### 0.1.3 Compatibility

This feature should not cause any compatibility problems.

#### 0.1.3.1 Backwards Compatibility

Existing applications will be able to upgrade DCE to include this feature, and run without modification. Client/Server applications on different hosts, where one is enhanced to use this feature, and the other is not, will be able to run without modification also. This is more an issue of coexistence, rather than compatibility, since the Unix Domain cannot be used across machines.

### 0.1.4 Installation and Configuration Impacts

When DCE is installed on a machine, the directory /usr/tmp/rpc must be created. If the directory already exists, then the directory should be wiped out and recreated. This is where the Unix Domain socket files will be created, and any files left hanging around this directory are stale, since it is assumed that all DCE applications are halted when a DCE installation upgrade takes place.

Configuration of DCE will be expanded. An administrator will be able to specify that a DCE instance only use ncacn\_unix\_stream as its supported protocol sequence. This will create a cell that cannot communicate with processes on different hosts, since Unix Domain sockets require that all interprocess communications take place on the same host.

#### 0.1.5 Performance and Storage Estimates

Performance of RPC calls will improve by 10-40% in the case where the client and server reside on the same host. Performance when the client and server exist on different hosts will not be affected in any way.

Storage estimates are not an issue. RPCD will store an extra endpoint for each server that does an rpc\_ip\_register() when support for Unix Domain sockets is allowed. A socket file will be created for each Unix Domain endpoint that a server registers, but this file is zero length.

The rpcclean utility (discussed previously in this document) can be run to remove stale socket files left lying around on the system. This will reduce the number of used inodes on the filesystem.

## Claims

1. A method for managing communication between a client process and a server process in a distributed computing environment, the client process residing on a host computer that is connected to a physical network having a transport layer and a network layer, comprising the steps of:

(a) when a remote procedure call is made by the client process, detecting whether a server process identified

process identified by the RPC is located on the host computer; and means responsive to the detecting means for using an interprocess communication mechanism of the host computer to facilitate the RPC.

5      13. A computer system for managing communication between a client process and a server process in a distributed computing environment, the client process residing on a host computer that is connected to a physical network having a transport layer and a network layer, said system comprising:

10            (a) means responsive to a remote procedure call being made by the client process, for detecting whether a server process identified by the remote procedure call is located on the host computer;  
              (b) means responsive to the server process being located on the host computer, for establishing an interprocess communication path between the client process and the server process without use of the transport and network layers of the physical network; and  
              (c) means for executing the remote procedure call.

15

20

25

30

35

40

45

50

55

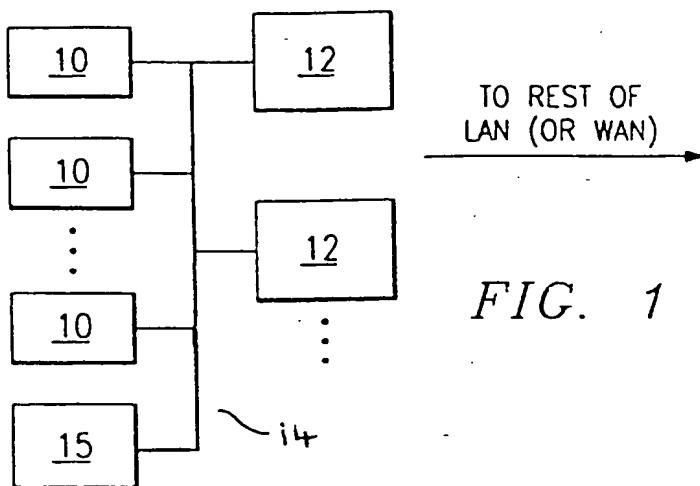
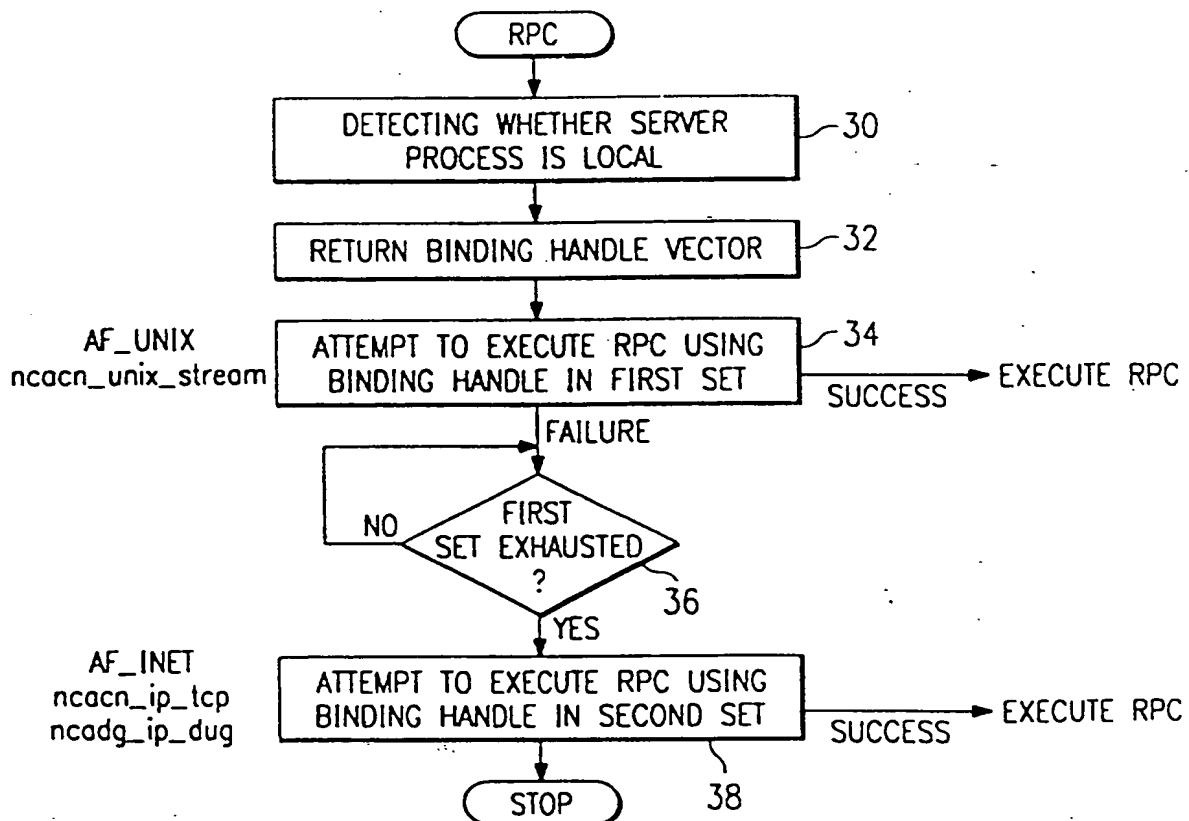


FIG. 3



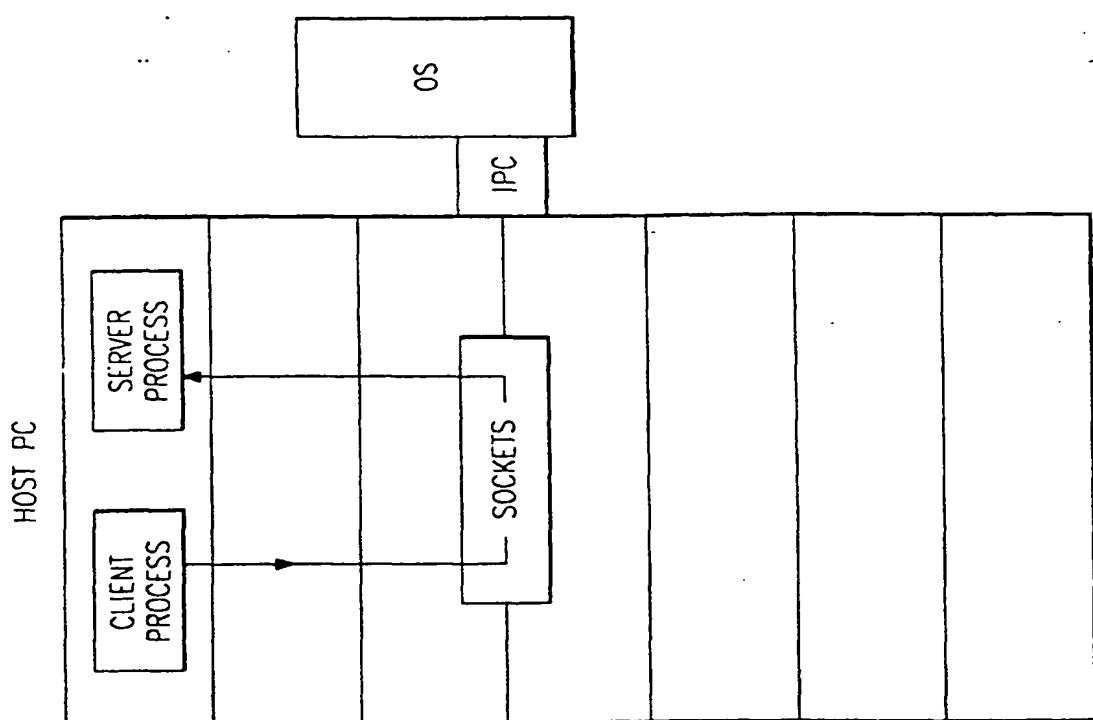
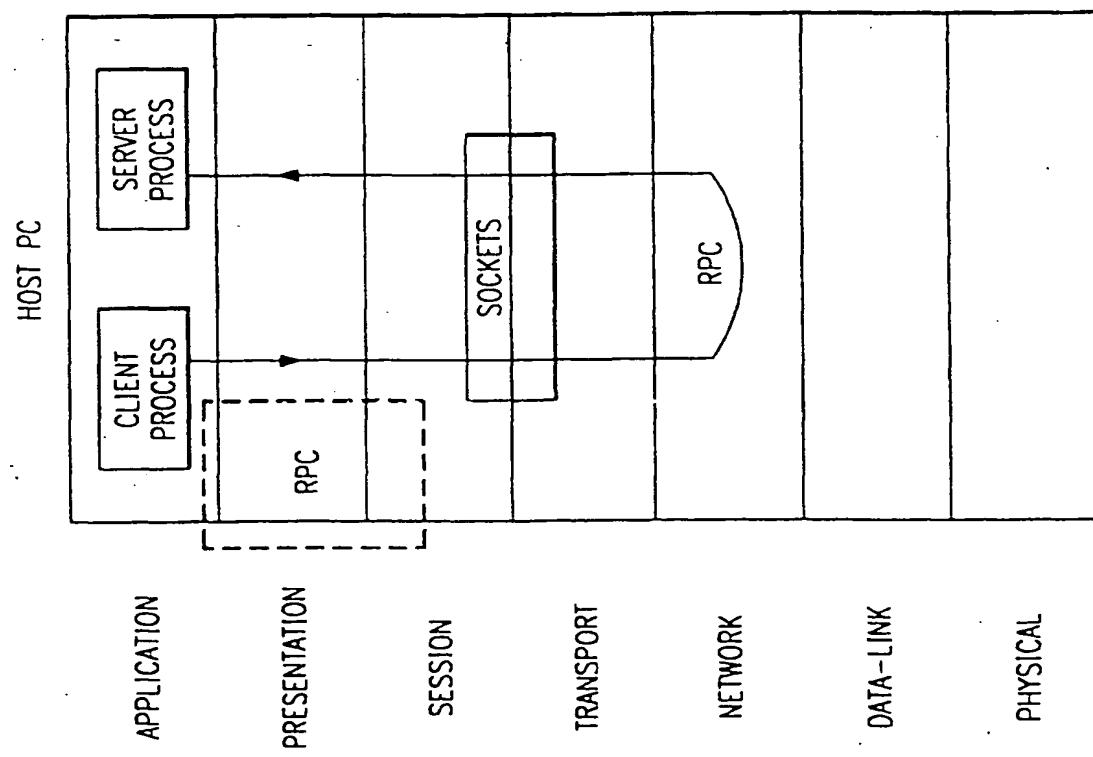


FIG. 2B

FIG. 2A  
(PRIOR ART)